CALIFORNIA INSTITUTE OF TECHNOLOGY
Computer Science Department
5105:TR:83

Memory Management in the Programming Language ICL

by

John Wawrzynek

November 20, 1983

Introduction

This paper presents the issues involved in implementing the programming language ICL and some of the details of the implementation, with special emphasis on aspects of the data management system. While the structures and algorithms presented here apply to all implementations of ICL, they are particularly relevant to the VAX implementation. This report is not intended to serve as an introduction to programming in ICL nor as a comprehensive guide to its implementation.

1.1 Introduction to ICL

The programming language ICL provides a coherent integration of many unique and experimental features. ICL is a higher level language than most popular programming languages in that it automatically manages many things which are conventionally left to the user. It is a strongly typed language like PASCAL, but is in some ways very similar to LISP. It provides automatic data swapping and has a garbage collector. The compiler and loader are *built in*, making the ICL system interactive . There are no explicit pointers and therefore no distinction between objects and pointers as in some other languages. In ICL the user is provided with a *work space* on disk as a place to hold data and programs between sessions.

An important aspect of ICL, particularly with respect to memory management, is it policy of subjective default. In other languages, such as PASCAL, when a data structure is modified all references or pointers to the data structure "see" the change, since the modification is made in place in the existing data structure. This results in side effects and unpredictable results. ICL avoids the problem by using *copy on write*. When a change to a data structure is made, a new copy of the structure is made with the change, and all old references to the original structure remain intact. The implementation is made efficient by copying only that part of the structure which is modified, with the remaining sections left to be shared among all references to the data structure. Through the use of small nodes to represent data structures, the maximum amount of sharing takes place.

Programming languages which use pointers to represent data structures and which are implemented on a Virtual Memory system suffer from performance degradation as program execution proceeds. This effect is because, after some time, the pieces which make up the data structures tend to be spread throughout memory. Since only a fraction of the address space of a program actually resides in memory at any one time, in order to access the pieces of a particular data structure, the system must swap in every page which contains a piece of the data structure. This excessive paging problem, which slows program execution, can be solved in one of two ways. The first solution is to compact data structures by a process which copies the pieces of each data structure into a physically close area in memory. This approach has been used in the programming language LISP. The second solution is one which is employed in the ICL implementation. The part of the address space which is allocated for data structures is kept small and a mechanism is supplied which efficiently moves data structures from main memory to the disk and back. This approach has been used by at least one other language in the past [Ingalls 78].

1.2 History of ICL

ICL grew out of a language for making pictures invented and implemented by Ron Ayres in 1974, while he was an undergraduate at Caltech. Later he became a graduate student and continued research in language processing. The results of his efforts was a general language processor with ICL as a sample language [Ayres 78]. ICL was intended to be a special language for integrated circuit design, thus the name Integrated Circuit Language (ICL). In fact, Ayres used ICL to implement one of the first working silicon compilers, RELAY [Ayres 79]. Whereas it does have some special features for integrated circuit design, ICL is a general and complete programming language.

The implementation of the language processor and ICL was initially done on a DEC-10 running TOPS-10 and later moved to a DEC-20 running TOPS-20. The first serious user of ICL, other than Ayres himself, was Dave Johannson, then a graduate student at Caltech. Johannson was also working on silicon compilation and used ICL to implement the Bristle Blocks silicon compiler [Johannsen 81]. Since then, ICL has been used extensively at Caltech to implement a variety of design aids including circuit simulators [Lin 83] and graphical systems for art-work generation [Ayres 83]. At USC-ISI ICL is used by the VLSI group and forms the basis of the MOSIS multi-project chip program.

With the proliferation of new machines on the market it was clear that ICL should be available on other machines. Also, the DEC-20 implementation had some limitations. Therefore, in the fall of 1981, it was decided to do a new implementation from scratch. The choice of the new host machine was a simple one. The DEC-20 was ruled out as the host because of its limited address space. The ideal host machine would have been one with the power of the DEC-20 and the size/cost/reliability of a personal computer. Nothing like that existed at the time.

The VAX offered moderate computing power, with many existing program development aids. It was a very established product and would surely be around for many years to come. The VAX had the extra advantage of providing a very large address space as a safety net against excessive memory use. Caution was taken from the beginning of the project, however, not to be fooled by the large virtual address space on the VAX because of potential loses in efficiency due to page faulting.

Working with a team of two, a paper design for the new run-time system and compiler was completed by the end of Spring 1982. By Spring 1983, a VAX/VMS ICL system was being used for production work within Silicon Compilers Incorporated. Improvements and enhancements are still in progress.

From the beginning of the project the VAX/ICL implementation had three primary goals:

1. To be compatible with the current DEC-20 implementation.

2. To serve as a basis for future ICL implementations.

3. To provide a performance improvement over the DEC-20 implementation.

The VAX/ICL project has been very successful in meeting the goals. Nearly all of the existing application programs have been installed in the VAX version of ICL with little or no modifications. Performance on the VAX is better than a hardware comparison suggests – runtimes are 1.5 to 1.7 times slower on VAX 11/750 compared to a DEC-20/60. Compile time is about 2 to 4 times slower.

ICL Organization

ICL provides an environment for developing, maintaining, and running programs along with facilities for storing and maintaining data objects. An important aspect of ICL is that it is interactive. Unlike some systems, there is not separate compiler and/or linker programs. Rather, the user of ICL normally uses the system by working at the terminal and typing ICL source text followed by the terminator (control-G). At this point the compiler processes the text and execution is transferred to the newly compiled code. This new code might simply define a function, in which case the only action is to store away a compiled version of the function for later use. However, if the new code has some top level action, something other than definitions, this new code is executed immediately. The new code might call a previously defined function which is loaded, linked and executed dynamically. Of course, one need not type all code directly to ICL, there is a function which will cause input to be taken from a file.

The interactive nature of ICL provides the user with the efficiency of compiled code along with the interactiveness of an interpreter. One benefit of this incrementally compiled environment is that the language itself may be used for debugging. A program bug or a problem causes the current state of execution to be saved and a new incarnation of ICL to be started. At this point ICL itself can be used to examine or set program variables and to resume execution.

In addition to an interactive compiler and debugger, ICL provides and maintains for the user a hierarchical directory system of data and code objects on the disk. The compiler stores away both compiled code and data objects in the hierarchy for later access. They are all stored in a large disk file called the VM file. The same file is used by the memory management system as a place to hold disked data structures when freeing up memory.

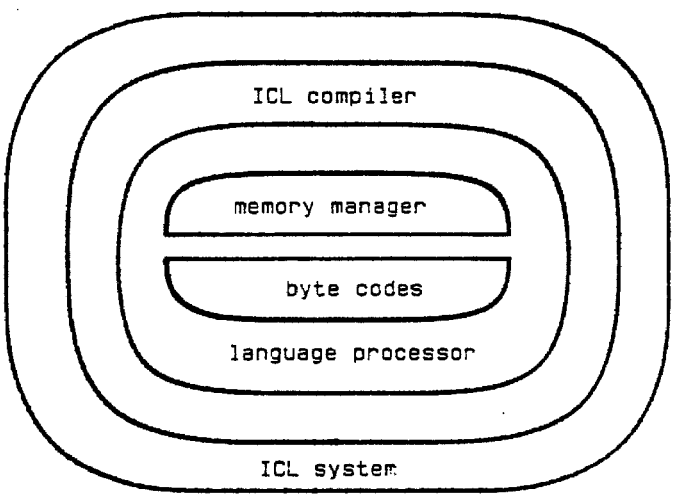

Figure 1: ICL Organization

Figure 1 shows how the ICL implementation is organized. At the core is the runtime system (RTS). The RTS provides low level support to the rest of the system. This piece has been implemented in machine

language for fast execution. The RTS is approximately 5,000 lines of VAX assembly language. The remaining pieces are all implemented in ICL. The ICL compiler is really just a customization of the general language processor with the grammars of ICL. The ICL system is a collection of programs which create the ICL working environment, including such things as debugging functions and functions for maintaining compiled code and data on disk.

The ICL runtime system

The runtime system has three major functions:

1. It provides low level support in the form of byte codes. Bytes codes come in two varieties. The first variety are those which implement the primitive functions of ICL. These primarily provide both a means to synthesize and analyze the basic data structures of ICL, such as STRINGS [1] and RECORDS. For example the record select byte code takes as arguments a pointer to a record and a select tag (integer) and returns a pointer to an element of the record. The second type of byte codes are those which provide support for the compiler.

2. The RTS performs memory management functions for ICL programs in an attempt to provide an endless supply of memory. Since all data structures in ICL are made from two word nodes, the sole interface between a running ICL program and the memory manager is the routine NEWNODE which returns a pointer to a free node for use by the program. The illusion of an endless supply of nodes uses both garbage collector and an automatic data object swap-out mechanism.

3. The last major function of the RTS is to provide an interface to the operating system, mainly IO.

3.1 Memory Layout

Main memory within the ICL implementation is divided into a few sections as is shown in figure 2. The first section is relatively small and contains the code of the RTS. Another relatively small section is the section called page space. In this space the RTS maintains a set of memory pages for use as IO buffers and as temporary memory during garbage collection when list space cannot be used. Code space is a large block of memory used to hold compiled ICL code as it is swapped in off the disk. List space is reserved for list nodes and is managed by the garbage collector. It is here that all data structures reside. List nodes are the stuff from which all ICL data structures are made.

[1] STRINGS in ICL are what some other languages call LISTS. A STRING *might* be a list of characters, but it is not limited to characters. A STRING can be a list of any object.

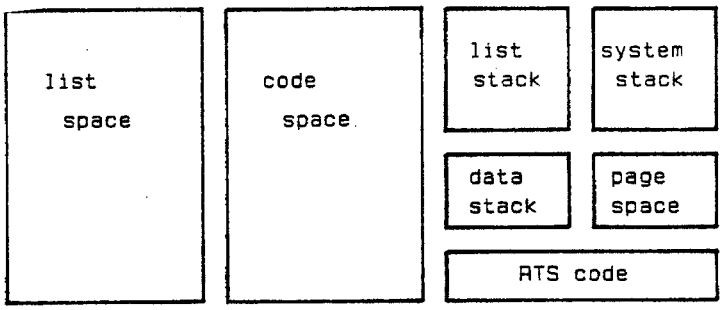

Figure 2: Memory Layout

Space for three stacks is provided. The system stack is used for CALL and RETURN and for passing parameters to the operating system. The list stack contains only pointers to valid ICL data structures – each entry on this stack is a root for the mark phase of the garbage collector. [2] The third stack does not contain roots for the marker and is normally used to hold non-pointer items. [3]

3.2 List Node Layout

All data structures in ICL are composed from list nodes. Each node is a two word entity as is shown is figure 3. Both WORD0 and WORD1 are 32 bit words (long word on the VAX). WORD0 is always a pointer to another node or a zero. WORD1 is either a pointer to another node or data. Pointers are restricted to be 24 bits or less in length, which leaves one free byte in WORD0 for use by the RTS as a tag. The tag is used to tell the RTS about the other fields in the node. Also, since WORD0 contains the address of another node, and nodes always lay on an eight byte boundry, the three low bits of WORD0 are without significance and free to be used internally by the RTS.

[2] See section 4.1.

[3] Early implementations of ICL used only one stack which resulted in some inefficiencies and undeterministic behavior in the garbage collection scheme. Since no explicit tag was used to differentiate between pointers and non-pointers (data) on the stack, a non-pointer occasionally would be misinterpreted as a valid pointer, and if it pointed into list space to a structure which would otherwise not be referenced, the structure would not be thrown away as garbage, as it should be.

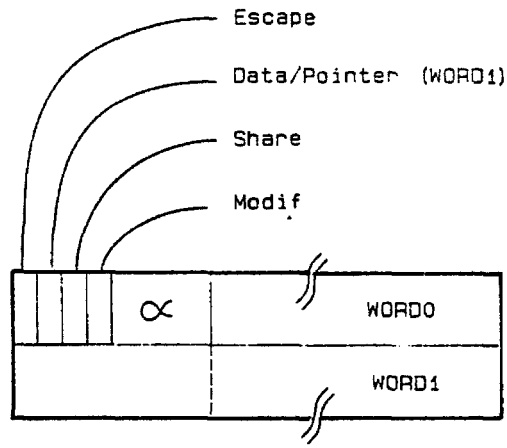

Figure 3: Node Layout

Referring again to figure 3, the tag byte can be divided into to two pieces. The right half α has different meanings depending on the application of the node. The bits of the left field always have the same interpretation - from left to right:

BIT7 is the *escape* bit - it is ZERO for the most common node types, those in ICL RECORDS, VARIANTS, and STRINGS. When *the escape bit is ONE the node is one of the several special nodes* differentiated by the α field. These nodes require special attention by the garbage collector and the swap-out mechanism.

BIT6 is the P/D bit. It is ZERO to indicate that WORD1 is a pointer and ONE when WORD1 is data.

BIT5 is the *share* bit, it is used by the marker during garbage collection. This bit is always clear when not being used by the garbage collector.

BIT4 is the *modif* or *dirty* bit, it indicates that a data structure has been modified since it was last written to the VM file.

In addition to the high byte of WORD0, the three low order bits of WORD0 are used by the RTS. BIT1 and BIT2 are used as mark bits in the garbage collector and swap-out mechanism. BIT0 is used to tell if WORD0 is a pointer or not. Non-pointer entities stored in WORD0 must not use the low three bits of WORD0.

3.3 Common Data Structures in ICL

Data types in ICL are divided into two groups. The primitive types include LOGICAL, INTEGER, REAL, SCALAR and BOOL. These types are implemented as one word, and always reside on the stack or as the WORD1 of a node.

The second group of data types are the non-primitive, or meta-types. These types are all composed of list nodes and include POINT, STRING, RECORD, VARIANT, DISK, and SUSFUNC.

3.3.1 POINTS

Consider the point (10.0,20.0), in ICL this is written `10.0#20.0`. It is implemented, as all other points, with two nodes. The first node's WORD1 contains the value 10.0, its WORD0 contains a pointer to the second node. The second node's WORD1 contains the value 20.0 and its WORD0 contains a zero. This is shown in figure 4. Note that the tag in both nodes is 40 hex to indicate that WORD1 is data.

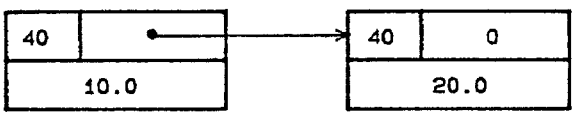

Figure 4: Point Layout

3.3.2 STRINGS

Strings in ICL can contain any type of object. For instance, the ICL source code for a type called SI which is a string of integers is

```
TYPE    SI =    { INT };.
```

A variable of this type can be made with the statement

```
VAR     SI1 =   SI;.
```

This declares a variable of type string of integers and gives it the name SI1. One way to assign values to SI1 is to simply enumerate its elements, for instance

```
SI1 := { 1; 2; 3; 5; 8; 13 };.
```

This statement generates a data structure with the implementation shown in figure 5. In this case, each element of the string holds one integer in its WORD1 and a pointer to the next element in its WORD0.

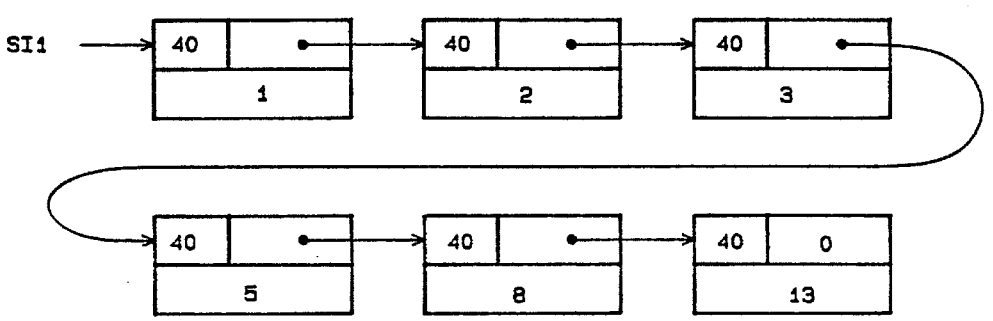

Figure 5: String of Integers

A slightly more complicated string than a string of integers is a string of points. This type which we will call SP is defined as follows

```
TYPE    SP =    { POINT };.
```

A particular string of points called SP1, may be created:

```
VAR     SP1 =   SP;
```

and a new point to help us form SP1.

```
VAR     ORIGIN = POINT;
        ORIGIN := 0#0;
```

Now for SP1

```
SP1 := { ORIGIN; 0#1; 1#1; 1#0; ORIGIN };
```

The resulting string is shown in figure 6. Note that the ORIGIN is shared by the first and last elements of the string.

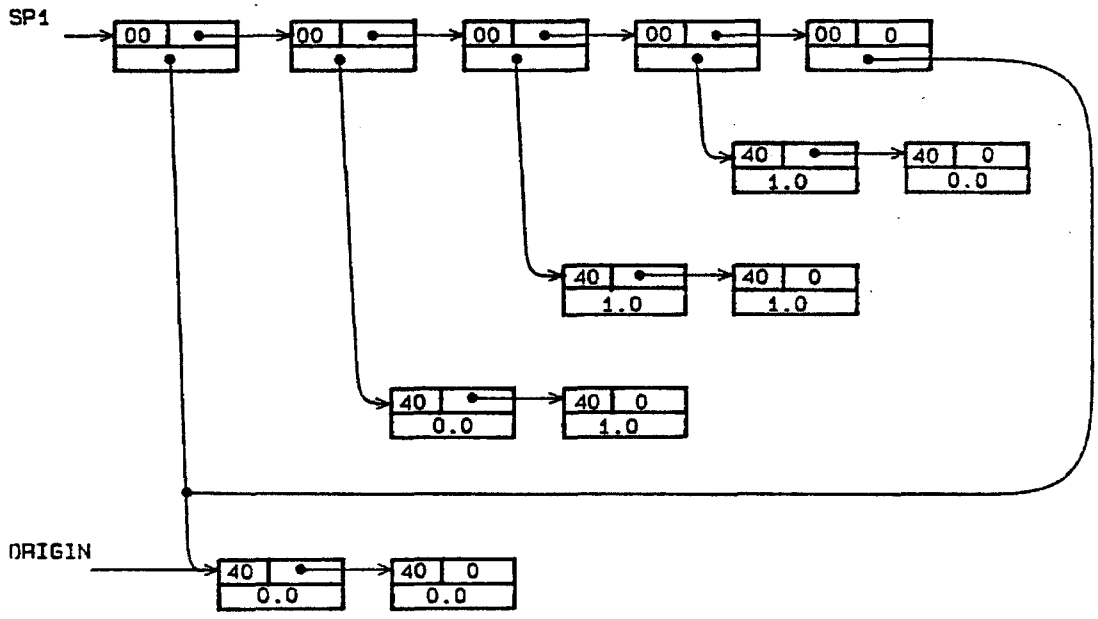

Figure 6: String of Points

In the two previous examples the α field is 0 for all the nodes. This is the internal representation for *left-append* nodes. *Left-append* nodes reside in a string *physically* in their correct *logical* order. But this is not always the case. Let us define two new strings SP2 and SP3 as a string of points,

```
VAR     SP2,SP3 =SP;,
```

and assign the first string a value.

```
SP2 := { ORIGIN; 1#1 };
```

We can now form the new string SP3 to as the concatenation of SP1 and SP2 with the ICL operator $$ follows

```
SP3 := SP1 $$ SP2;.
```

With this assignment SP3 *logically* is the string { 0#0; 0#1; 1#1; 1#0; 0#0; 0#0; 1#1 } but physically is represented by the construction shown in figure 7. A string node with the α field set to 2 hex has its WORD0 point to the first part of the string (SP1) and its WORD1 point to the second part of the string (SP2). We will not discuss the pros and cons of this representation for strings here, [4] suffice it to say that four types of nodes are used in strings – *left-append, right-append, concatenation,* and *reverse.* A *left-append* node is a node whose WORD1 points to or contains an element which logically appears to the left of the

[4] Interested readers should refer to [Ayres 78].

string which is pointed to by WORD0. A *right-append* node is similar to a *left-append* node but its element logically belongs at the end of the string pointed to by WORD0. The *reverse* node indicates that the elements in the string pointed to by WORD0 logically belong in reverse order.

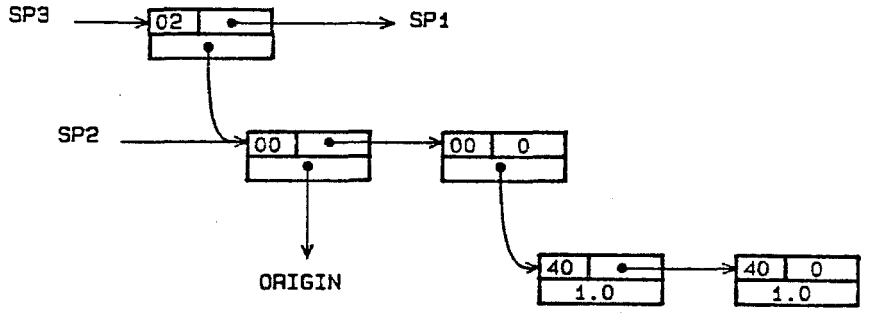

Figure 7: Concatenation of Strings

3.3.3 RECORDS

Let us define a new data type for wires. A wire can be represented as a string of points, along its center path, and a width. For this we will use a RECORD with two components.

```
TYPE    WIRE = [ WIDTH: REAL PATH: SP ];
```

Now we can define an instance of a wire which uses SP1 from the last section.

```
VAR     W1 =    WIRE;
        W1 :=   [ WIDTH: 2.0 PATH: SP1 ];
```

The implementation of W1 is shown in figure 8. The implementation of RECORDS is very similar to that of STRINGS with the following notable exceptions:

- The elements of STRINGS always contain or point to objects of the same type, whereas RECORDS may be composed of objects of all different types.
- The α field in RECORDS is used to differentiate between the various components of the RECORD. Each component is assigned a unique integer value. Nils are not represented explicitly.

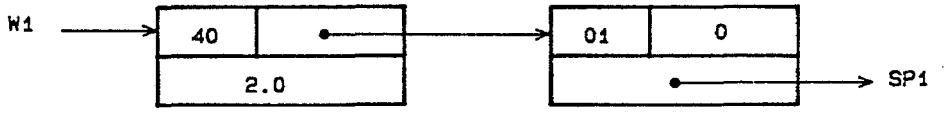

Figure 8: Representation for WIRE RECORD

3.3.4 VARIANTS

With VARIANTS we define objects which may take on more than one type. The concept is very similar to VARIANT RECORDS in PASCAL. For example, we will define a THING to be either a WIRE or a single POINT.

```
TYPE    THING =EITHER W=WIRE P=POINT ENDOR;
```

Instances of the type THING can be objects of type WIRE or POINT. Which type of object a particular instance is, is called the *state* of the VARIANT. Variants are implemented with one node. WORD0 contains a number for the state of the VARIANT and WORD1 contains the value of the VARIANT.

3.3.5 DISK Objects

A DISK object in ICL is a candidate to be swapped out by the memory manager when memory gets low. Any pointer type of data object in ICL can become a diskable object simply by declaring it as such. For example suppose we declare a new type called PICTURE as follows

```
TYPE    PICTURE = ...;
```

PICTURE is no doubt a collection of various shapes. Now if we wish to make instances of PICTURE which may be swapped out to the disk we declare a new type DISK_PICTURE

```
TYPE    DISK_PICTURE= DISK PICTURE;
```

Note the keyword DISK on the right hand side of the declaration. Any instance of a DISK_PICTURE can be completely interchanged with an instance of a PICTURE, the RTS will automatically take care of moving the object back and forth between memory and the disk if necessary.

When a diskable object is declared the compiler creates two new definitions, which would appear as follows if they were declared by the user:

```
LET     DISK_PICTURE BECOME PICTURE BY ...
DEFINE DISK( P:PICTURE)=DISK_PICTURE: ...  ENDDEFN
```

The first is an *implicit function* – it is a function without a name which is applied by ICL when it is needed in order to make the types compatible. For example we can define a function called DRAW which takes as its argument an object of type PICTURE and renders it on the terminal

```
DEFINE DRAW( P:PICTURE ) ...  ENDDEFN,
```

and then call it with an object of type DISK_PICTURE, say DP1

```
DRAW(DP1);
```

The compiler will compile in a call to the implicit function described above in order to satisfy the type of the argument needed by DRAW. At run-time the implicit function checks to see if DP1 is currently *core resident* or not. If the object is on the disk, it is swapped in at this point.

The second declaration automatically made by the compiler for disk objects is one which takes the non-disk form of an object and returns the disk form. This function is used by the user to signal to the RTS which objects to consider for swap-out. This facility provides the user with the mechanism to decide the best place to fragment data structures into diskable pieces.

There is a good deal of *behind the scenes* mechanism to handle disk objects, which we will discuss in length later. For now, it is important to note that disk objects are implemented with the use of one extra node inserted before the non-disk version of the object. WORD0 of the disk node points to the non-disk object (or is nil if the object is not swapped in). WORD1 holds an index into a table of disk addresses. The

swap-in mechanism uses the disk address to locate the object on the disk. WORD1 is nil if a disk address for this object has not yet been established.

3.3.6 SUSFUNCS

Suspendable functions (susfuncs) in ICL are a way to treat code as data. That is, a piece of compiled code is encapsulated, passed around as data and executed at a later time, thus the name *suspendable function.*

Susfuncs are useful in simulation programs. Imagine an ICL function which is declared as follows:

```
DEFINE  SIMULATE( C:CIRCUIT S:STOPPER R:READER W:WRITER ) ...
```

along with the following types:

```
TYPE    CIRCUIT     =       . . .
        STOPPER     =       // BOOL \\;
        READER      =       // (CIRCUIT) \\;
        WRITER      =       // (CIRCUIT) \\;
```

The type CIRCUIT is a representation for a circuit whose behavior is to be emulated (perhaps a list of transistors and nodes). STOPPER, READER, and WRITER are all type declarations for suspendable functions. STOPPER returns a boolean value when executed. Both READER and WRITER accept one argument of type CIRCUIT and return *nothing* when executed. The body of the function SIMULATE is as follows:

```
DEFINE  SIMULATE( C:CIRCUIT S:STOPPER R:READER W:WRITER ):
        WHILE -<*S*>   DO      <*R*>(C)
                               .
" Code to perform function of circuit based on argument C "
                               .
                               .
                               <*W*>(C);       END;
ENDDEFN
```

The simulation is terminated when executing the STOPPER yields a true value. If executing S yields a false value, a new step of simulation is performed by 1) executing the READER to provide new input values to the circuit, 2) computing new output values based on the input and finally, 3) providing the WRITER with the updated circuit. The new circuit values are available to the WRITER to be processed. The implementation details of the three susfuncs can be delayed until run time when the SIMULATE function is invoked. For example, we define several versions of the STOPPER and as long as they all return a boolean value when executed, each is perfectly acceptable. The simplest susfunc for the STOPPER is a piece of code which asks the user if he wishes to continue each time the STOPPER is executed. Let's call this STOPPER ASK.

```
VAR     ASK =  STOPPER;
        ASK := //      DO WRITE('Continue [Y/N]?');
                       GIVE TTYIN='Y' \\;
```

Another method of stopping the simulation is to specify a particular number of iterations N to be performed. For this we define a function which when invoked returns an instance of a STOPPER initialized such that it executes N times before returning true.

```
DEFINE CYCLES( N:INT ) = STOPPER:
        // { N; } (N ::= -1;) < 0 \\
ENDDEFN
```

The iteration count is held as state by the STOPPER returned by the function CYCLES and is decremented each time the susfunc is executed. The function SIMULATE can now be invoked with one of the available STOPPERS:

```
SIMULATE( CIRCUIT, ASK, ... );
```

or with CYCLES:

```
SIMULATE( CIRCUIT, CYCLES(100), ... );
```

The second invocation causes the circuit to be simulated for 100 cycles.

Susfuncs in ICL, as with all other data types, are implemented using list nodes. A susfunc is a list of nodes, the last one of which is a special node called a LINK node which provides the mechanism for getting the address of the code for the body of the susfunc. The remaining nodes are used to store the data associated with the susfunc. It is important to note that two different kinds of data are associated with susfuncs. The first type is data which the susfunc holds as local state, as is the case of the iteration count N in the susfunc returned by CYCLES. The other is data passed to the susfunc as an argument at execution time as in the case of CIRCUIT for the susfuncs of the type READER and WRITER. The second type of data is only available at the time the susfunc is executed and is not stored with the data structure representing the susfunc. The body of the susfunc is compiled into normal code space as a function which takes as parameters all of the arguments which are to be passed to the susfunc at execution time, along with the susfunc's local state.

The Garbage Collector

The RTS provides a node from list space, for the user program, with the NEWNODE routine. NEWNODE simply takes a node off the free-list and returns it to the program. If the free-list is exhausted, however, the garbage collector is invoked to restock it. The garbage collector restocks the free-list by three methods, 1) reclaiming the nodes from unreferenced data structures 2) swapping out some diskable data structures and reclaiming their nodes and 3) if necessary, acquiring new memory pages for use as list nodes.

The ICL garbage collector couples a simple *mark and release* algorithm [Knuth 68] with a list environment containing diskable objects. Since the garbage collector needs to examine every node in memory during the marking process, it takes this opportunity to determine which objects should be swapped out.

Once invoked, the garbage collector attempts to create a free-list at least 10% of the size of the total number of active nodes in memory. This is accomplished by initiating a *mark and release* phase, which *may* cause disk objects to be swapped out to the disk. After this phase, if the 10% quota has not been reached, more memory pages are acquired and turned into list nodes.

ICL maintains an invariant that an object in the VM file never refers to an object in memory. This invariant ensures that the garbage collector does not have to follow pointers on the disk – a very slow process. A separate process is invoked manually to garbage collect the VM file.

An alternative method to explicitly swapping data structures is to let list space grow very large and have the virtual memory hardware and software on the host machine handle moving data from main memory to the disk and back when necessary. Pointer structures inherently do not fit well into this approach since list structures inherently do not reside in local neighborhoods in main memory, but rather are spread throughout list space. Therefore this method would result in excessive page faulting. ICL keeps the size of list space small by handling its own data swapping which reduces the amount of page faulting, and speeds up garbage collection.

4.1 Disk Node Management

To understand the basic garbage collection algorithm it is necessary to see how list space is organized, particularly with respect to disk nodes.

At the time the garbage collector is invoked, which may be any time, all references to list structures are on the list stack, in registers, or from other list structures. This means that the roots for the garbage collector are the list stack and the registers. Every node in list space can be reached by following pointers from the roots. All list nodes which can be reached from the roots of the garbage collector without passing through a disk node are said to belong to the *initial segment.*

Three kinds of disk nodes exist:

a. A brand new disk node created by ICL is called an *enfant* disk node. It is created by the DISK function which takes in a structure and a returns the diskable version of the same structure. Enfant disk nodes have no DNUM and their incore field is always defined.

b. A *modif* (modified) disk node is one which must ultimately be written out. It has a DNUM. Its INCORE field (WORD0) is defined.

c. An *established* disk node has a DNUM and its version on the disk is up-to-date. Its INCORE field may or may not be defined.

All disk nodes start out as *enfants* and if they survive, spend their whole lives going back and forth between *modif* and *established.* In practice, once nodes arrive in the *established* state, most never make it back to the *modif* state, since their associated structures are not modified. An *enfant* disk node goes to a *modif* type if a disk structure which references it gets swapped out, and in the process is assigned a DNUM.

All disk objects in the *modif* of *established* state are referenced from a special list called ALDSK. Actually ALDSK is not a single list but a collection of lists emanating from a hash table. Disk nodes are hashed by their WORD1 (DNUM), which is an index into a table of disk addresses. The hash table ALDSK provides a quick way to find a disk node given its DNUM. This facility is used during by the swap-in process. But not every disk node is referenced from ALDSK. It is possible that a new disk node in the *enfant* state could be garbage collected away before it ever needs to be written to the disk, in which case it will never be assigned a DNUM.

Another special list for disk nodes is GRADES. This list of disk nodes is arranged in the order in which they will swap-out, if it becomes necessary. It is rebuilt during each garbage collection and is used by the next garbage collection for the swap-out phase.

4.2 Basic Algorithm

The basic garbage collection algorithm breaks into six phases. In this discussion we will assume the presence of a routine which recursively marks list nodes, and a routine which takes a list structure and writes it out to the VM file.

Phase 1 — The initial segment is marked (all nodes which can be reached from the registers and stack including disk nodes, but NOT the diskable structures themselves.)

Phase 2 — The GRADE list is used to mark the nodes of disk objects. This process continues until the maximum number of nodes allowed to be resident in memory is reached. This maximum number is called MAXMRK - and is somewhat analogous to a working set size in a virtual memory system. Once MAXMRK has been reached the remaining disk objects on the GRADES list are written out to the VM file, for later retrieval if necessary. As each disk object which is staying in memory is marked, its grade is computed which determines its position in the new GRADE list and hence ultimately its priority for being swapped out during the next garbage collection. Each remaining disk object is marked with a second mark bit called MARK', swapped out, and the incore field of the disk node cleared.

Phase 3 — All disk nodes without the MARK bit or the MARK' bit set are removed from ALDSK because some disk objects may have ceased to be referenced since the last garbage collection. The ALDSK list itself is marked.

Phase 4 — The new GRADE list is marked and assigned into the GRADE list.

Phase 5 — This is the sweep phase. A linear scan of list space is preformed. All nodes which have not been marked are returned to the free list. The mark bit is cleared in all nodes.

Phase 6 — If the size of the free list is not at least 10% the size of the space occupied by the active nodes, then more memory is acquired and turned into new nodes which are added to the free list.

In phase 2 of the garbage collector, a grade is computed for each disk object which is not being moved to the disk. The current scheme for computing the grade is to simply use the number of nodes which will be freed when the disk object is released. Disk objects which free up more nodes when released are ranked to swap to the disk before those which release less. This is not always equal to the total size of the disk object because some of its substructures may be shared with an object which is not being released.

4.3 The Markers

Two separate markers are used by the garbage collector. One is used for marking the *inital segment* and disk objects which will be remaining in memory. The other marker is used to mark disk objects before they are swapped out of memory.

The function of both markers is to recursively traverse a list structure depositing a mark in each node. The marker finds its way through a list structure by using the node tag, which indicates which word of the node is a pointer and which is immediate data. Of course, pointers are followed and data is not. The markers return with the number of nodes marked. This quantity is used to determine the *grade* of disk objects.

The disk object marker, in addition to setting the mark bit in each node it encounters, provides some additional functions. As the marking process proceeds, if the marker encounters a node with its mark bit

already set it will set the share bit. The share bit is sensed by the swap-out process. In this manner, the system maintains shared data across disking sessions.

The other function of this marker is to propagate the modif or dirty bit from any point in the data structure to the disk node. In this manner, disk objects which have not been modified since last being written to the disk need not be written again. The modif bit is set by one operator in ICL called the @ operator, which is used to override the *subjective default* or copy-on-write quality of ICL [Ayres 78].

4.4 The Swap-out and Swap-in Processes

The swap-out process is invoked from phase 2 of the garbage collector. It is passed a candidate disk structure for the disk. Only if the disk structure is in the ENFANT or modified (MODIF) state is it actually written to the disk. On the disk list structures are represented as a linear stream of bytes, where pointers are replaced by what they point to. Pointers to other disk nodes are represented on the disk by the disk node's DNUM.

Special attention is paided to nodes which have their share bit set. A set share bit indicates that the node is referenced from more than one other node. In order to preserve sharing through the swap-out/swap-in process, shared nodes are treated as symbols. That is, when a shared node is encountered it is written to the disk, and is also assigned a number. On subsequent references to the same node, only its number is written to the disk and not any other nodes which it points to. In this manner those parts of a structure which are shared reside only once on the disk. Upon swap-in, the structure headed by the share node is swapped in and also placed in a symbol table according to its number. Subsequent references to shared nodes on swap-in are handled by looking up the number of the symbol in the table and pointing directly to the previously swapped in shared section of the data structure. This preservation of sharing allows swapping of list structures containing cyclic references.

The swap-in process is called from a running ICL program when the program references a data structure which is currently not in memory. The swap-in process rebuilds the data structure from its linear representation on the disk using nodes from the free list.

Conclusion

The current implementation of ICL on the VAX runs with the VMS operating system. Currently, work is proceeding to bring up ICL with Berkeley Unix V4.2. ICL will be ported to personal machines as they become more powerful. We believe that ICL will be more effective on a personal computer than it has been on time sharing systems.

Many performance enhancements to ICL are currently underway. These changes include better file buffering algorithms for the ICL VM file and improving the code generator. A major improvement to ICL, currently in redesign and implementation, is a better swap-out metric for disk objects. Empirical evidence shows that the current scheme of ranking disk objects purely on size is not an efficient one. A *least recently used* algorithm would increase the performance of the memory management system.

A possible future enhancement to ICL includes an automatic diskizing of data structures. This enhancement would replace the current system where it is the responsibility of the user to define which objects are candidates for the disk. Though at times the explicit diskable declarations are an inconvenience, this responsibility may be best left to the user – for the user has intimate knowledge of the data structures and the dynamics of the program execution. In practice, users find the extra declarations a small price to pay in return for the services provided.

The ICL language has helped to increase the productivity of users by providing high level system services and managing data through the use of disk object. The management of disk objects is important now and will continue to be so as the magnitude of applications grow, particularly in the field of Integrated Circuit Design.

References

[Ayres 78] Ayres, R.
A Language Processor and a Sample Language,
Doctoral Dissertation,
Computer Science, Caltech, Pasadena, Calf. 1978.

[Ayres 79] Ayres, R.
A Hierarchical Use of PLA's,
IEEE Computer Society,
16th Design Automation Conference Proceedings,
San Diego, California, 1979.

[Ayres 83] Ayres, R.
VLSI Silicon Compilation and the Art of Automatic Microchip Design,
Prentice Hall, 1983.

[Ingalls 1976] Ingalls, D. H.
The Smalltalk-76 Programming System Design and Implementation
Proc. of the 5th Annual ACM Sym.
on Principles of Programming Languages,
Tuscon, Arizona, January 1978.

[Johannsen 81] Johannsen, D.
Silicon Compilation, Doctoral Dissertation,
Computer Science, Caltech, Pasadena, Calf. 1981.

[Knuth 68] Knuth, D.E.
The Art of Computer Programming, Vol. 1
Addison Wesley, Reading, 1968.

[Lin 83] Lin, T., Mead, C.
Signal Delay in General RC Networks with Application to Timing of Digital Integrated Circuits,
Technical Report # 5089, Computer Science,
Caltech, Pasadena, Calf. 1983.